

Mobile Agents for Reliable Migration in Networks

DongChun Lee¹, Byeongkuk Jeon², and Yongchul (Robert)Kim³

¹ Dept. of Computer Science Howon Univ., Korea
ldch@sunny.howon.ac.kr

²Dept.of Office Automation WonJu Nat'l College, Korea

³Embedded System Team LGIS RD Center, Korea
robertkim@lgis.com

Abstract. Mobile agents are autonomous objects that can migrate from one node to other node of a computer network. Due to communication nodes failures, mobile agents may be blocked or crashed even if there are other nodes available that could continue processing. To solve it, we propose a scheme with the path reordering and backward recovery to guarantee migration of mobile agents in networks.

1 Introduction

Mobile agents are autonomous objects that can migrate from node to node of a computer network and provide to users which have executed themselves using databases or computation resources of hosts connected by network. To migrate the mobile agent, it is needed a virtual place so-called the mobile agent system to support mobility [1]. Many prototypes of mobile agent systems have been proposed in several different agent systems such as Odyssey [2], Aglet [3], AgentTCL, Mole[5], and so forth. However, most systems are rarely ensuring its migration for a fault of communication nodes or a crash of hosts to be caused during touring after a mobile agent launches. That is, when there are some faults such as a destruction of the nodes or the mobile agent systems, mobile agents may be destroyed to block or orphan state even if there are available other nodes that continue processing. Because of the autonomy of mobile agents, there is no natural instance that monitors the progress of agent execution.

2 Previous Mobile Agent For Migration

Mobile agents are migrated autonomously according to the relevant routing schedule, and then accomplished their goals. Figure 1 depicts how a node repository can use for implementation instead of transaction message queue for agents. Assume that an agent moves from a node to the consecutive node along the path $N_1 N_2 \dots N(k-1) N_k$ (where N_i is a network node, H_i is a host, R_i is an agent repository). As an agent may visit the same node several times N_i and N_j ($1 \leq i, j \leq k$),

$j \leq k$) may denote the same or different nodes. Assume further that an agent is stored in a repository when it is accepted by the agent system for execution. Except N_k , each other node performs the following sequence of operations on Transaction T_i such as Get (agent); Execute (agent); Put (agent); Commit. Get removes an agent from the node's repository. Execute performs the received agent locally. Put places it on the repository of the host that will be visited the right next time. Three operations are performed within a transaction and hence consisted of the atomic unit of work.

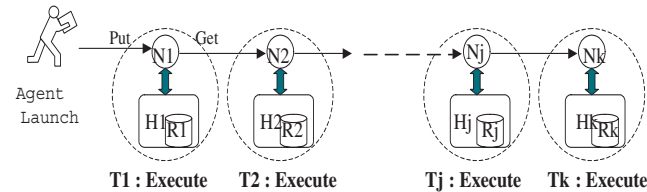


Fig.1. A migration path of a mobile agent.

In Fig. 1, we assume to happen a failure in a particular node N_i within the migration path of the mobile agent. Though the node N_i of the host H_i of node N_i lives, the agent can't migrate. Inversely, though the node N_i can be communicated with the previous node N_{i-1} , the agent can't occasionally migrate if the host H_i does not operate in the agent system. In the above cases, the agent never arrives by the last node N_k . the agent at previous host H_{i-1} needs to receive user's assertion. In the worse case, if a shared host on the multiple agents launched occurs to crash on executing (launching), the agents will block or destroy.

3 Proposed Scheme

We describe an scheme for the agent system to support reliable migration of mobile agents even if it dose happen some failures hosts on the cluster of computer networks. The scheme adapts 'fault types' such that agents are not able to migrate more continuously.

3.1 Reordering of The Whole Path

The mobile agent is impossible to migrate to the destination node by the fault of node or host crash. Fig. 2(a) supposes that there is a migration path corresponding with an agent's routing schedule and some faulty nodes, such as N_3 , N_4 , and N_7 . An agent migrates and executes from node N_1 to N_2 sequentially, but it is blocked at the host of node N_2 until the node N_3 is recovered. If the node N_3 dose not recovered, the agent may be orphan or destroyed by the particular host. To solve this situation is for the agent to skip the fault node N_3 that includes on the migration path and move the address of node N_3 to the last one of the migration path. Hence, the node N_2 successfully connects the next other node N_4 without any fault. As the same method is also applied to other nodes, the

agent's migration path has reordered. This solution changes the previous migration path by connecting with normal nodes except that some nodes have the particular fault. Afterward, the agent retries to connect each certain fault node after it waits for the time-stamp to assign by the mobile agent system. If the certain fault node is recovering during the time-stamp, the agent succeeds to the migration. Otherwise, the address of the fault node will be discarded. Fig. 2(b) shows that all migration path for the mobile agent is changed by this scheme.

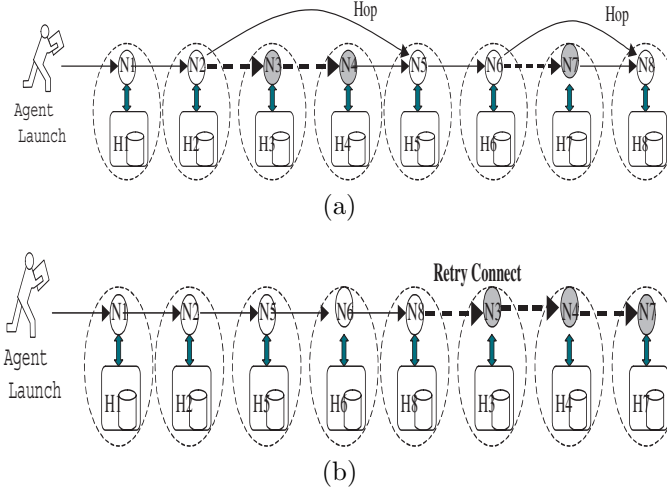


Fig.2. Faults of nodes on a migration path and reordering

The Path Reordering executes connecting to communicate with the mobile agent system. If the agent doesn't connect the destination node, it succeeds with connecting the right next node, after the failed address is moved to the last one of routing table and that will be retried to connect about the node. When it does reconnect each failed destination address, it does wait for the time-stamp to be assigned by the mobile agent system to connect. If it does fail again, it does ignore this address, and repeat to connect the next fault node. And then, if it does adapt to more than twice times failed node, a mobile agent may be occurred loophole for connection. So it limits to retry. Although it is connected, if each host of nodes errors the mobile agent system, it is adapted equally. In this way, algorithm 1 offers automatically to reorder the migration path.

Algorithm 1. Path Reordering

```

For each agent's routing-table {
  extract a target address and fail_checked information;
  if(no more a target address)backward multicastes 'Agent_Fire'
    signal to successful_target nodes;
  if(is it a fail_checked_address) {
    wait the agent during some system_timestamp;
    try to connect Socket to the address;
  }
}

```

```

    if (success){ call goAgent;
                  exit;
    } else { notify to user the address is unavailable;
            ignore the address;
    } }
else if (not a fail_checked_address) {
    try to connect Socket to the destination node;
    if (success) { call goAgent;
                  exit;
    } else { notify to user;
            move the current failed_address to last in the routing-table;
            set the fail_checked information;
    } } }

```

3.2 Backward Recovery

In Fig. 3, we suppose that migrated agents execute autonomously at the host H5. If the host H5 of node N5 crashes, all agents at that host are blocked or destroyed. To prevent it, when an agent migrates after it ends its job at a previous host, the agent's clone leave equally itself at that host. Then, the clone is unconditionally waiting for an acknowledge signal 'ACK' that reaches from the next host. If the signal 'ACK' doesn't reach within the time-stamp from the next host H5, the cloned agent waiting for at the host H4 has automatically activated since it resolves to any hindrance. Then, it is passed by the node N5 and hops to the next node N6. If the migrated agent faults at the host H6 on execution, it will be repeated the same method. However, the running agent in a host H5 is destroyed by being clashed, and at the same time if the prior node N4's host occurs succeeding fault, the cloned agent has already copied the prior host H3 wakes up and re-runs. This is so-called Backward Recovery.

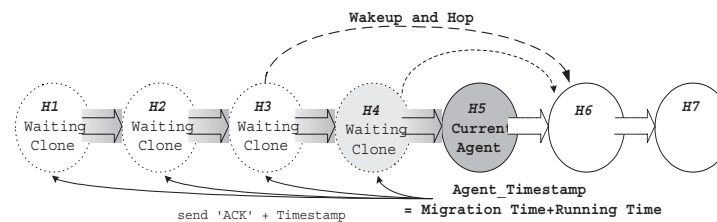


Fig.3. An example of Backward Recovery

The Backward Recovery is as follows: The agent system leave the clone of the agent being already passed at all hosts from source to current node and each clone is waiting during it's own time-stamp. Here in, the time-stamp of each clone is maximum at source, the next will be less reflecting the migration and execution time of the prior, and so forth. Since an agent is launch, it's time-stamp accumulates informing to every clone of the prior hosts it's own moving and running time before it depart for the current host. Therefore, clones are

waiting during the time-stamp. Each clone spontaneously revives and redoes the path reordering regarding that host as clashed if none received any signal from next host. At the last node's host, the agent system broadcasts a signal 'Agent_Fire' to be destined all copied of the agent except the faulty nodes and failed hosts until reaching the destination.

Algorithm 2.A Backward Recovery

```

Waiting Clones Check {
  for each slepted_Clone
    if (empty a Clone_timestamp) notify to user;
    call wakeup Clone; }
goAgent {
  send the agent;
  wait the agent's 'ACK' signal during send_timestamp;
  if ( 'ACK') { clones the agent;
    call sleepAgent; }
  else call wakeupAgent;
    } arriveAgent { send 'ACK' to the previous_node;
    execute the agent;
    } sleepAgent {
      for each cloned_agent {
        add agent_timestamp to system_time-stamp;
        add the agent to the slepted_list;
        sleep the agent;
      }
    }
} wakeupClone {
  for the slepted_list find a cloned_agent;
  remove it from the sleep_list;
  if('Agent_Fire')remove the cloned_agent;
  else {
    move the current failed_address to last in the routing-table;
    set the fail_checked information;
    call arrangePath at the algorithm 1;
  }
}

```

4 An Implementation

Our scheme is implemented in the JAvA Mobile Agent System (JAMAS) that we developed. As shows in Fig. 4, the JAMAS consists of Graphic User Interface, Agents Mobile Service component, Agents Execution Environment component, and Agents Repository to provide the naming transparency of agents. In addition, it may be executing one more systems within a host

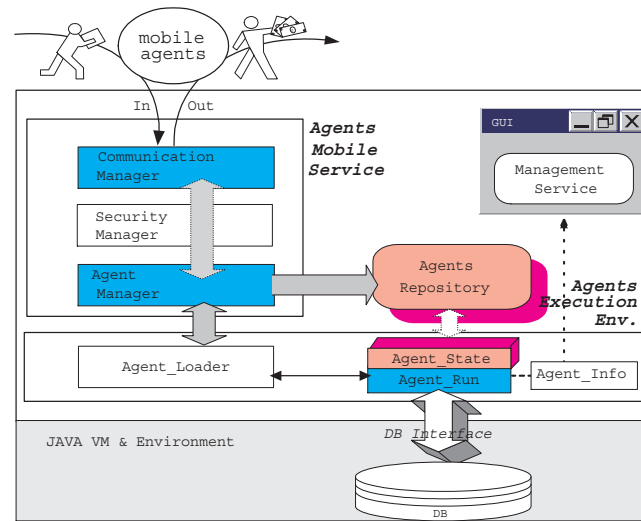


Fig.4.The architecture of JAMAS

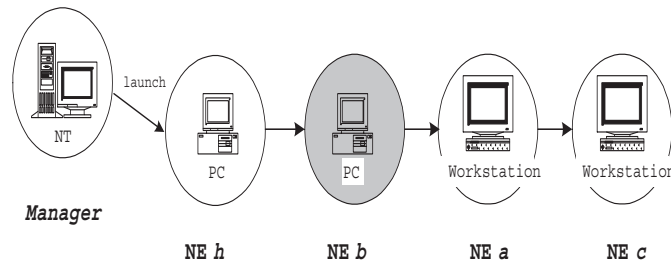


Fig.5. A routing path having a fault NE b

We show to experiment with an agent which manages some NE (network elements). The following figures show the progress that the sample agent as a role of MIB (Management Information Base) browser is migrated and executed according to the routing schedule. Fig. 5 depicts the routing path of the sample agent such as NEh, NEb, NEa, NEc, and we assume faulty at the host NEb. The network manager fetches the prepared agent and specifies routing addresses of it to migrate. So, clicking the 'Go' button on the manager's window to launch it, the agent starts on a tour to get the MIB information of each NE on behalf of the network manager.

Fig.6 shows screen shots of results of the mobile agent. The agent tracer GUI shows what nodes have faulty and how to migrate continuously in the network. The executed agent at the host IP address 172.16.53.21 of the first node NEh does migrate to the second node NEb. Due to a particular fail, the agent has been hopped and migrated at the third node NEa. On completing the execution at the last node NEc, it results information of reconnection to the faulted node NEb on the reordered path. Finally, Fig.7 realizes execution of the agent at each NE. Fig. 7 (a) as a screen capture of the host NEh, shows hopping by connection

failure at the next NEb after the launched agent normally progresses. That is, due to fail the host, the agent passes to next one. Thereafter, Fig. 7(b), (c) capture executing of the agent at the hosts NEa, NEc. Then it is adapted to the our scheme. Therefore, the agent has toured for all nodes having no faults before that it does re-connect with the fault nodes.

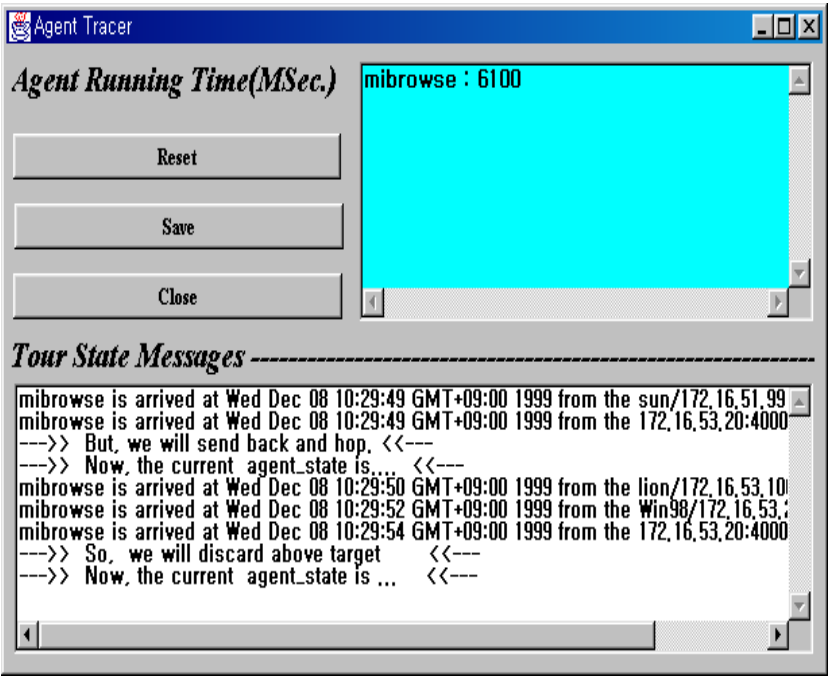
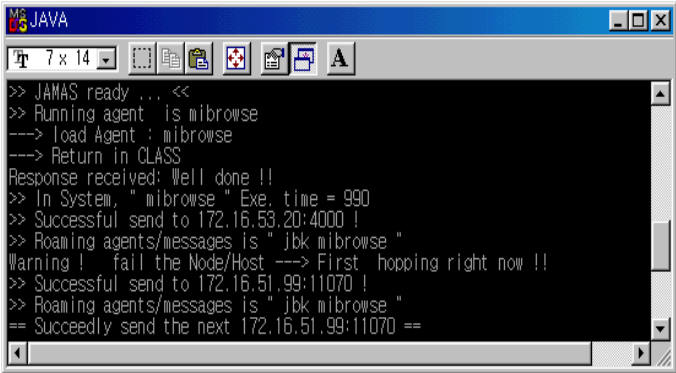
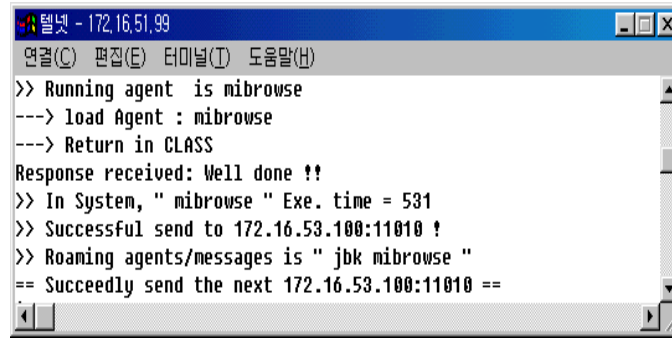


Fig.6.Agent_Tracer GUI



(a) AScreen shot of executing at the NEh

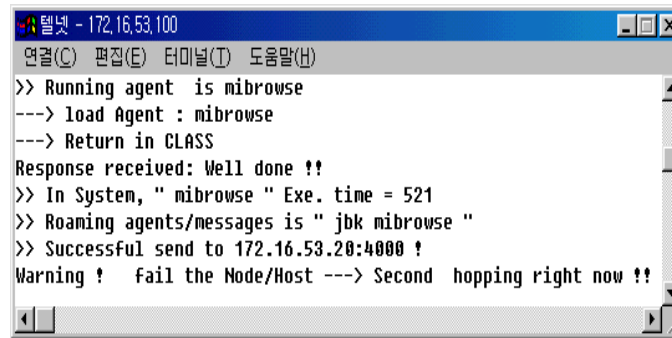


```

텔넷 - 172.16.51.99
연결(C) 편집(E) 터미널(T) 도움말(H)
>> Running agent is mibrowse
---> load Agent : mibrowse
---> Return in CLASS
Response received: Well done !!
>> In System, " mibrowse " Exe. time = 531
>> Successful send to 172.16.53.100:11010 !
>> Roaming agents/messages is " jbk mibrowse "
== Succedely send the next 172.16.53.100:11010 ==

```

(b) screen shot of executing at the NEa



```

텔넷 - 172.16.53.100
연결(C) 편집(E) 터미널(T) 도움말(H)
>> Running agent is mibrowse
---> load Agent : mibrowse
---> Return in CLASS
Response received: Well done !!
>> In System, " mibrowse " Exe. time = 521
>> Roaming agents/messages is " jbk mibrowse "
>> Successful send to 172.16.53.20:4000 !
Warning ! fail the Node/Host ---> Second hopping right now !!

```

(c) A screen shot of executing at the NEc and attempting migration of the second at the NEb

Fig.7.Fault-tolerable execution of a mobile agent at each NE

5 Conclusions

We discuss a fault-tolerable scheme with the path reordering and backward recovery to ensure the migration of mobile agents in networks. The proposed scheme not only affords to avoid faults of communication nodes or hosts of mobile agents, but also affects to agents' life span.

References

1. OMG, "Mobile Agent Facility Interoperability Facilities Specification(MAF)", OMG
2. General Magic, "Odyssey", and URL: <http://WWW.genmagic.com/agents/>
3. IBM,"The Aglets Workbench", URL: <http://www.trl.ibmco.jp/aglets>
4. Robert S.G, "AgentTCL: A Flexible and Secure Mobile-Agent System", TR98-327, Dartmouth Col. June 1997
5. J.Baumann,"A Protocol for Orphan Detection and Termination in Mobile Agent Systems", TR-1997-09,Stuttgart Univ. Jul.,1997.